Course Booklet for LINUX Internals Programming

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

Is it C or Linux?

By Emertxe

Version 2.2 (December 18, 2014)

Contents

Chapter 1

OS & LINUX Basics

1.1 Expectations out of this module

Notes:

1.2 OS Basics

An operating system, or OS, is a software program that enables the computer hardware to communicate and operate with the computer software. OS are found on almost any device that contains a computer with multiple programs from cell phones and video game consoles to super computer and web servers.

It makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

Operating systems can be classified as follows:

multi user
multi user : Allows two or more users to run programs at the same time

multiprocessing
Supports running a program on more than one CPU

multitasking
Allows more than one program to run concurrently

multithreading
Allows different parts of a single program to run concurrently

real time
Responds to input instantly

Notes:

1.3 Linux Basics

Linux is inspired by the Unix operating system which first appeared in 1969, and has been in continous use and development ever since. Many of the design conventions behind Unix also exist in Linux and are central to understanding the basics of the system.

Unix was primarily oriented towards the command line interface, and that legacy is carried on in Linux. Thus, the graphical user interface with its windows, icons and menus are built on top of a basic command line interface. Furthermore, this means that the Linux file system is structured to be easily manageable and accessible from the command line.

Notes:

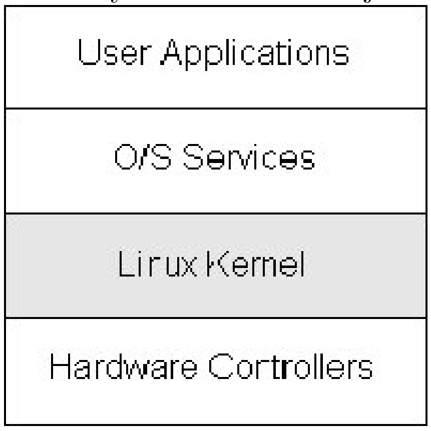
1.3.1 Linux Properties

- Linux is free
- Linux is portable to any hardware platform
- Linux was made to keep on running
- Linux is secure and versatile
- Linux is scalable
- The Linux OS and most Linux applications have very short debug-times

Notes:

1.4 Components of LINUX

Linux System into the Major Subsystem.



Notes:

1.4.1 User Application

They are the set of applications in use on a particular Linux system. Examples include a word-processing application and a web-browser.
Notes:

1.4.2 OS Services

These are services that are typically considered part of the operating system (e.g. windowing system, command shell)
Notes:

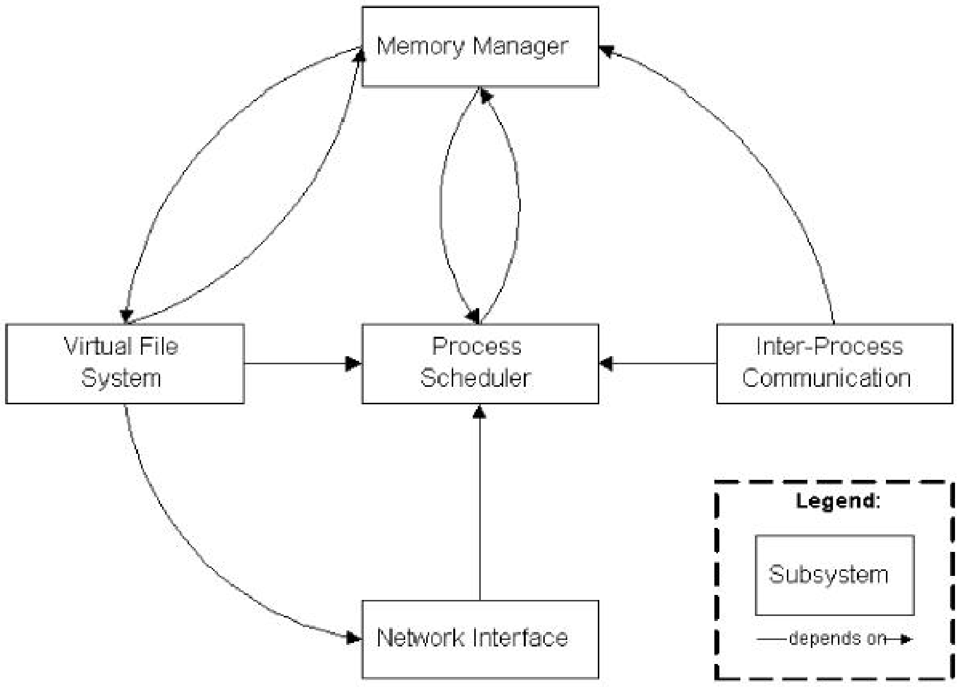
1.4.3 Linux Kernel

Notes:

1.4.4 Hardware

Notes:

1.5 Overview of the Kernel Sturcture



Notes:

1.5.1 Process Scheduler(SCHED)

Process scheduling is the heart of the Linux operating system. The process scheduler has the following responsibilities:

- allow processes to create new copies of themselves
- determine which process will have access to the CPU and effect the transfer between running processes
- receive interrupts and route them to the appropriate kernel subsystem
- send signals to user processes
- manage the timer hardware
- clean up process resources when a processes finishes executing

Notes:

1.5.2 Memory Manager (MM)

Another important resource that's managed by the kernel is memory. For efficiency, given the way that the hardware manages virtual memory, memory is managed in what are called pages (4KB in size for most architectures). Linux includes the means to manage the available memory, as well as the hardware mechanisms for physical and virtual mappings.

Supporting multiple users of memory, there are times when the available memory can be exhausted. For this reason, pages can be moved out of memory and onto the disk. This process is called swapping because the pages are swapped from memory onto the hard disk.
Notes:

1.5.3 Virtual File System (VFS)

The virtual file system (VFS) is an interesting aspect of the Linux kernel because it provides a common interface abstraction for file systems. The VFS provides a switching layer between the SCI and the file systems supported by the kernel.
Notes:

1.5.4 Network Interface (NET)

The Linux network system provides network connectivity between machines, and a socket communication model.
The Linux network system provides two transport protocols with differing communication models and quality of service. These are the unreliable, message-based UDP protocol and the reliable, streamed TCP protocol. These are implemented on top of the IP networking protocol.
Notes:

1.5.5 Inter-Process Communication (IPC)

The Linux IPC mechanism is provided so that concurrently executing processes have a means to share resources, synchronize and exchange data with one another. Linux implements all forms of IPC between processes executing on the same system through shared resources, kernel data structures, and wait queues.
Notes:

1.6 Kernel Designs

Most older operating systems are monolithic, that is, the whole operating system is a single a.out file that runs in 'kernel mode.' This binary contains the process management, memory management, file system and the rest. Examples of such systems are UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.

The alternative is a microkernel-based system, in which most of the OS runs as separate processes, mostly outside the kernel. They communicate by message passing. The kernel's job is to handle the message passing, interrupt handling, low-level process management, and possibly the I/O. Examples of this design are the RC4000, Amoeba, Chorus, Mach, and the not-yet-released Windows/NT.
Notes:

1.7 LINUX File System

1.7.1 Ext2 File System

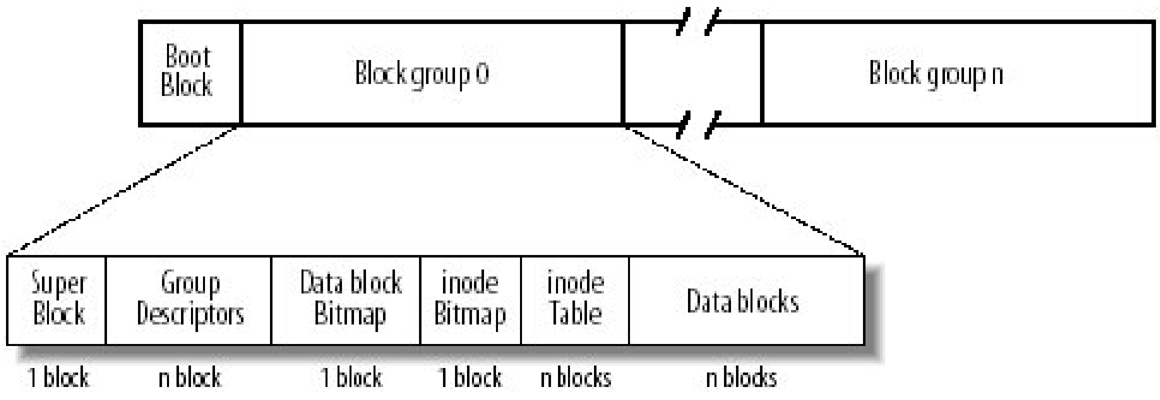
The first versions of Linux were based on the Minix filesystem. As Linux matured, the Extended Filesystem (Ext FS), The Second Extended Filesystem (Ext2) was introduced in 1994.

1.7.2 FEATURES:

- Block fragmentation
- Access Control Lists (ACL)
- Logical deletion
- Journaling

Notes:

1.7.3 Ext2 Disk Data Structures



Notes:

1.7.4 File Types

File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

Notes:

1.7.5 /proc File System

The /proc filesystem contains a illusionary filesystem. It does not exist on a disk. Instead, the kernel creates it in memory. It is used to provide information about the system (originally about processes, hence the name).

```
$ /proc/cpuinfo- gives CPU related information
$ /proc/version
```

Notes:

Chapter 2

System Calls

2.1 User and Kernel spaces

System memory is divided into two. Kernel space and User space.

Kernel Space is where Kernel executes and provide its services.User space is memory locations where user processes (everything other than kernel) runs.

One of the job of Kernel is to manage individual user processes within this space and to prevent them from interfering with each other.
Notes:

2.2 System Calls

Kernel space can be accessed by user process only through the use of system calls. System calls are request by an active process for a service performed by the Kernel, such as I/O or process creation.

I/O is any programs operation or device that transfers data to or from a CPU and to or from a peripheral device (such as disk drives, Key Board, mouse and printers).

System Calls can be grouped into 5 major categories:

- Process Control
- File Mangement
- Device Management
- Information Maintainance
- Communication

Notes:

2.2.1 System Calls and Library functions

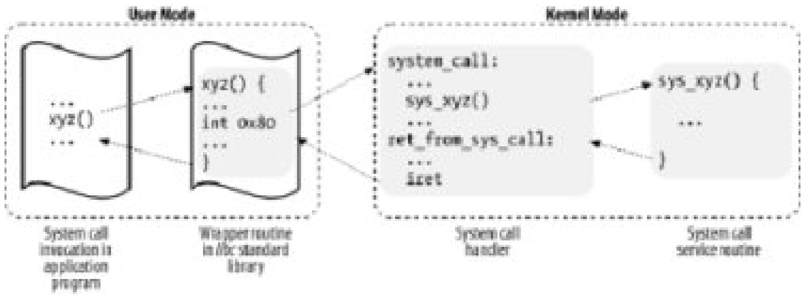
A library function is an ordinary function that resides in a library external to your program.

A call to a library function is just like any other function call.The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the functions code, which typically resides in a loaded shared library.

A system call is implemented in the Linux kernel.When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes. A system call isnt an ordinary function call, and a special procedure is required to transfer control to the kernel.

Notes:

2.2.2 System Calls in Detail


User Mode
Kernel Mode
System call invocation in application program
Wrapper routine in libc standard library
System call handler
System call service routine

```
...
xyz()
...
```

```
xyz() {
...
int 0x80
...
}
```

```
system_call:
  ...
  sys_xyz()
  ...
ret_from_sys_call:
  ...
  iret
```

```
sys_xyz() {
  ...
}
```

Notes:

2.2.3 Tracing System Calls

strace:

Strace is a tracing utility that intercepts and records the system calls that are called by a process and the signals that are recieved by a process. The name of each system call, it's arguments, and its return value are printed to STD_ERR or to the specified file.

Eg : strace ls

Notes:

2.3 System Call Examples

2.3.1 File Related System Call

- open
- close
- read
- write
- lseek
- fcntl
- dup
- dup2

Notes:

2.3.2 Time Related System Calls

Unix time, or POSIX time, is a system for describing points in time, defined as the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds.

- time
- gettimeofday
- localtime
- ctime
- strftime
- Date +%s – command

Notes:

2.3.3 High Precision Sleep

- sleep
- nanosleep

Notes:

2.4 Practice - I

W.A.P to copy a file to another using system calls.

2.4.1 Prerequisites

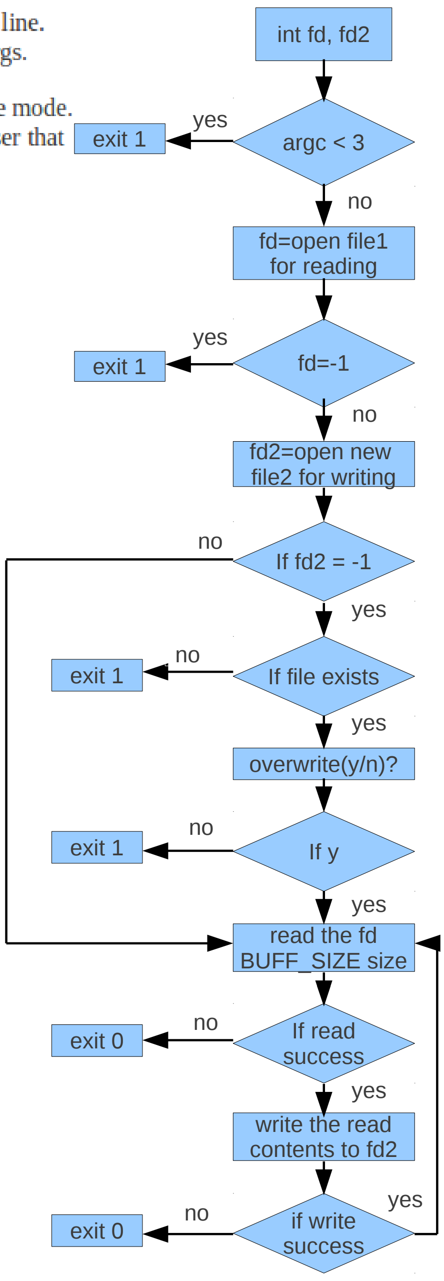
1. File related system calls

2.4.2 Objective

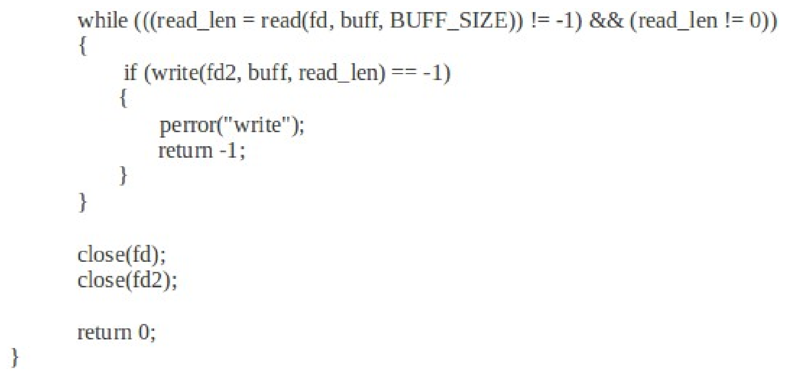
1. Familiarising commonly used file related system calls.

2.4.3 Algorithm Design

1. Pass the file names through command line.
2. Check the number of command line args.
3. Open the first file in read mode.
4. Open the second file in write exclusive mode.
5. If the file already exists, prompt the user that user needs to overwrite the file or not.
6. If user selects 'no', exit the program.
7. Else overwrite the file.
8. If file is not existing, open a new file.
9. Copy the contents of file1 to file2.
10. Close all the open files after copying.



```
int main(int argc, char *argv[])
{
    int fd, fd2, read_len;
    char choice;
    char buff[BUFF_SIZE];

    if (argc < 3)
    {
        printf("Usage : ./a.out <source file> <destination file>\n");
        return 1;
    }

    if ((fd = open(argv[1], O_RDONLY)) == -1)
    {
        perror("open");
        return -1;
    }

    if ((fd2 = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0666)) == -1)
    {
        if (errno == EEXIST)
        {
            do
            {
                printf("File Exist: Do you want to overwrite (y/n) ?\n");
                scanf("%c", &choice);
                getchar();
                if (choice == 'n' || choice == 'N')
                    return 0;
                else if (choice == 'y' || choice == 'Y')
                {
                    if ((fd2 = open(argv[2], O_WRONLY | O_CREAT, 0666)) == -1)
                    {
                        perror("open");
                        return -1;
                    }
                    else
                        break;
                }
            } while (1);

        }
        else
        {
            perror("open");
            return -1;
        }
    }
```



```
        while (((read_len = read(fd, buff, BUFF_SIZE)) != -1) && (read_len != 0))
        {
              if (write(fd2, buff, read_len) == -1)
              {
                    perror("write");
                    return -1;
              }
        }

        close(fd);
        close(fd2);

        return 0;
}
```

2.4.4 Dry Run

2.4.5 Practical Implementation

1. File Operations.

2.5 List of Assignments

(Id) / Date	Assignment Topic
( ) ____	Implement cp command along with -p option
( ) ____	Implement wc command for files as well as stdin
____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	

Chapter 3

Processes

3.1 Processes

3.1.1 What is a Process?

Running Instance of a Program is called a Process.

The most fundamental concept in an Operating system is the process.Linux is a timesharing system in which processes compete for system resources. Every process is scheduled to run for a period of time, called time slice or quantum. Processes are rescheduled either when its time slice expires or when it blocks for I/O.

A process competes for system resources and CPU with other processes. Once it gains control of the CPU, it runs a program within its own address space. The execution of a process is independent of the execution of the others, and it cannot read or write data into the address space of other processes.

A process consists of the executing program code, a set of resources such as open files, internal kernel data, an address space, one or more threads of execution and a data section containing global variables.
Notes:

3.1.2 Process Versus Program

Program is an inactice, static entity consisting of a set of instructions and a set of data.

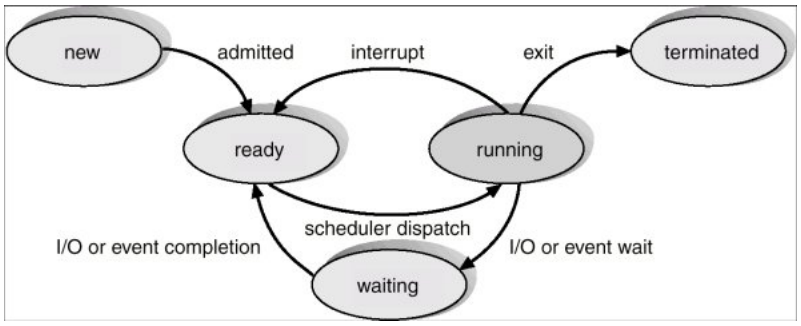
Process is an instance of a program running in a computer. Process is a dynamic entity scheduled and controlled by the Operating System.
Notes:

3.2 Process States

- New State: The process being created
- Running State: A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant
- Blocked (or waiting) State: A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens
- Ready State: A process is said to be ready if it use a CPU if one were available. A ready state process is runable but temporarily stopped running to let another process run
- Terminated state: The process has finished execution

Notes:

3.2.1 Process state Transitions



Notes:

3.3 Schedular

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling.
When more than one process is runable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm. All the scheduling algorithms are discussed in the process scheduling chapter.
Notes:

3.3.1 Operations

- create a process, adding it to the set of ready processes
- dispatch one of the ready processes to the processor
- if process timed out, removing it from the processor and returning it to the set of ready processes
- if process blocked, removing it from the processor and adding it to the set of blocked processes
- wakeup a blocked process, moving it into the set of ready processes
- destroy a process

Notes:

3.4 Process Descriptor/Structure

In the kernel, the process descriptor is a structure called task_struct, which keeps track of process attributes and information. All kernel information regarding a process is found there.
Notes:

3.4.1 State Field

TASK_RUNNING

The process is runnable; it is either currently running or on a runqueue waiting to run . This is the only possible state for a process executing in user-space; it can also apply to a process in kernel-space that is actively running.
Notes:

TASK_INTERRUPTIBLE

The process is sleeping (that is, it is blocked), waiting for some condition to exist. When this condition exists, the kernel sets the process's state to TASK_RUNNING. The process also awakes prematurely and becomes runnable if it receives a signal.
Notes:

TASK_UNINTERRUPTIBLE

This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly. Because the task does not respond to signals in this state, TASK_UNINTERRUPTIBLE is less often used than TASK_INTERRUPTIBLE.
Notes:

TASK_STOPPED

Process execution has stopped; the task is not running nor is it eligible to run. This occurs if the task receives the SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal or if it receives any signal while it is being debugged.
Notes:

TASK_ZOMBIE

The task has terminated, but its parent has not yet issued a wait4() system call. The task's process descriptor must remain in case the parent wants to access it. If the parent calls wait4(), the process descriptor is deallocated.
Notes:

3.4.2 Process ID

Every process has a process-ID, which is a positive integer. At any instant this is guaranteed to be unique. Every process but one has a parent. The exception is process 0, which is created and used by the kernel itself, for swapping.

- process-ID : Positive integer that uniquely identifies this process.
- parent-process-ID : Process-ID of this process's parent.
- process-group-ID : Process-ID of the process-group leader. If equal to the process-ID, this process is the group leader.

Notes:

3.5 Basic Process Management

3.5.1 Viewing active process (Commands)

PS command

Notes:

PS fields

- S: process status (R: runnable, S: sleeping, T: suspended, Z: zombie)
- UID: effective user ID of the process
- PID: ID of the process
- PPID: ID of the parent process
- TIME: amount of CPU time used
- CMD: command
- NI: nice value
- C: percentage of CPU time used by the process
- PRI: priority of the process
- SZ: size of the process in KB
- TTY: the controlling terminal
- WCHAN: Memory address of the event the process is waiting for

Notes:

3.5.2 Getting Process ID in C-Program

- getpid()
- getppid()

Notes:

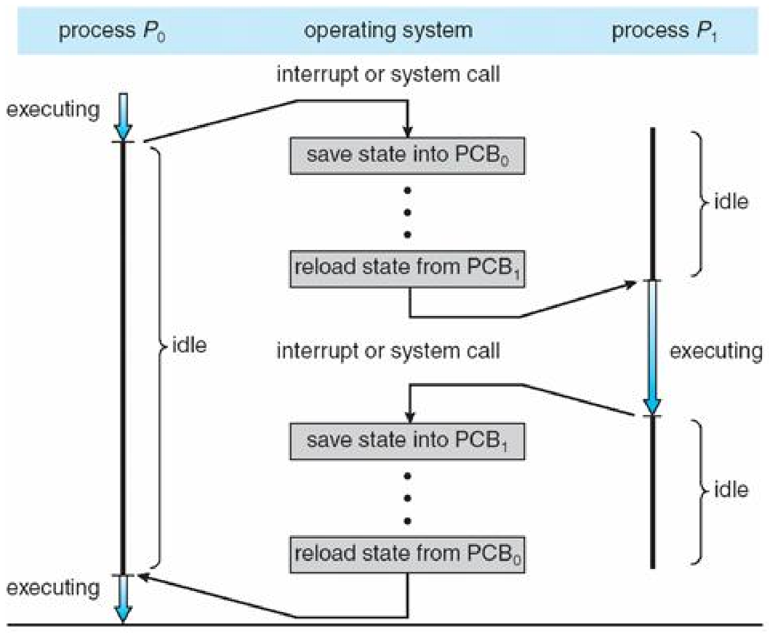
3.5.3 Foreground and background process and killing a process

- bg : Starts a suspended process in the background
- fg : Starts a suspended process in the foreground
- jobs: Lists the jobs running
- pidof: Find the process ID of a running program
- top: Display the process that are using the most CPU

Notes:

3.5.4 Context switch

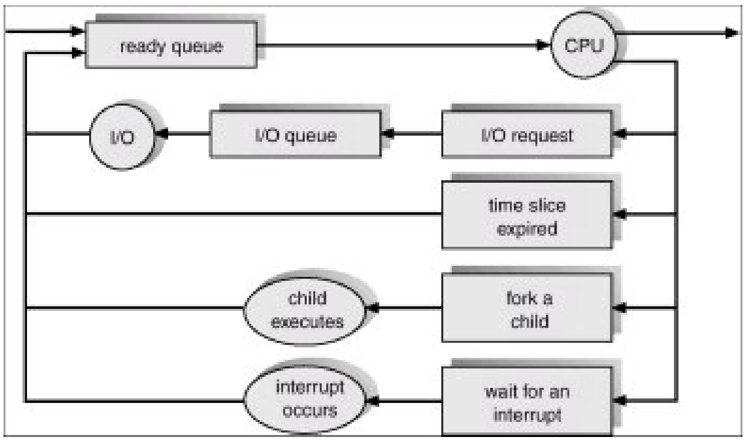
A context switch occurs when the kernel transfers control of the CPU from an executing process to another that is ready to run. The kernel first saves the context of the process. The context is the set of CPU register values and other data that describes the process' state. The kernel then loads the context of the new process which then starts to execute.Context-switch time is overhead; the system does no useful work while switching.



Notes:

3.5.5 Process Scheduling Queues

- Job queue : set of all processes in the system
- Ready queue : set of all processes residing in main memory, ready and waiting to execute
- Device queues : set of processes waiting for an I/O device



Notes:

3.6 Creating a Process

3.6.1 system() Function

The system function is used to issue a command. Execution of your program will not continue until the command has completed.

```
    system("wc -l");
```

Notes:

3.6.2 fork()-Creating a New Process

pid_t fork (void);
The fork function creates a new process.

If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process.

The specific attributes of the child process that differ from the parent process are:

- The child process has its own unique process ID.
- The parent process ID of the child process is the process ID of its parent process.
- The child process gets its own copies of the parent process's open file descriptors. However, the file position associated with each descriptor is shared by both processes.
- The elapsed processor times for the child process are set to zero.
- The child doesn't inherit file locks set by the parent process.
- The child doesn't inherit alarms set by the parent process.
- The set of pending signals for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process).

Notes:

3.6.3 vfork()

vfork() is designed to be used in the specific case where the child will exec() another program, and the parent can block until this happens. A traditional fork() required duplicating all the pages of the parent process in the child - a significant overhead.

While fork makes a complete copy of the calling process's address space and allows both the parent and child to execute independently, vfork does not make this copy. Instead, the child process created with vfork shares its parent's address space until it calls _exit or one of the exec functions. In the meantime, the parent process suspends execution.

Notes:

```
int main()
{
    pid_t pid;
    if (pid = fork())
    {
        ---- do something ---- /* parent */
    }
    else
    {
        ---- do something ---- /* child */
    }
    return 0;
}
```

3.6.4 exec()-Replacing Process Image

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process ID.
- Session and process group membership.
- Real user ID and group ID, and supplementary group IDs.
- Pending alarms.
- Current working directory and root directory.
- File mode creation mask.
- Process signal mask.
- Pending signals.
- Elapsed processor time associated with the process.
- Signals that are set to be ignored in the existing process image are also set to be ignored in the new process image. All other signals are set to the default action in the new process image.

Notes:

```
int main()
{
    printf("calling ls -l\n");
    execl("/bin/ls", "ls", "-l", NULL);
    printf("exec fails\n");
}
```

3.6.5 exec family

- execvp
- execlp
- execve
- execl
- execle

Notes:

3.6.6 Shortcomings of fork() and exec()

```
pid_t pid;
if((pid = fork()) == 0)
{
    execl("/bin/tail", "tail", "messages", NULL);
}
else
{
    wait(&status);
    ----- do some other work -----
}
```

Notes:

3.6.7 Copy-On-Write (COW)

Copy on Write is an important technique in the linux kernel memory management. The basic idea is to prevent the creation of unnecessary copies of structures when creating new processes.

Pages in the parent's region are not copied to the child's region until needed. Both parent and child can read the pages without being concerned about sharing the same page. However, as soon as either parent or child writes to the page, a new copy is written, so that the other process retains the original view of the page.
Notes:

3.7 Wait Family of System Calls

- wait
- waitpid
- waitid
- wait3
- wait4

pid_t wait(int *status) : used to wait until any one child process terminates.

wait (&status) If the exit status value of the child process is zero, then the status value reported by waitpid or wait is also zero. You can test for other kinds of information encoded in the returned status value using the macros mentioned in man page of wait. These macros are defined in the header file sys/wait.h.
Notes:

3.8 Special Case of Processes

3.8.1 Zombie Process

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the process that started the (now zombie) process to read its exit status.When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, at which stage the zombie is removed.After the zombie is removed, its process ID and entry in the process table can then be reused.
Notes:

3.8.2 Orphan Process

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running.In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically. Even though technically the process has the "init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists.
Notes:

3.9 Practice - I

Write a program to show parent child relationships by using fork.

3.9.1 Prerequisites

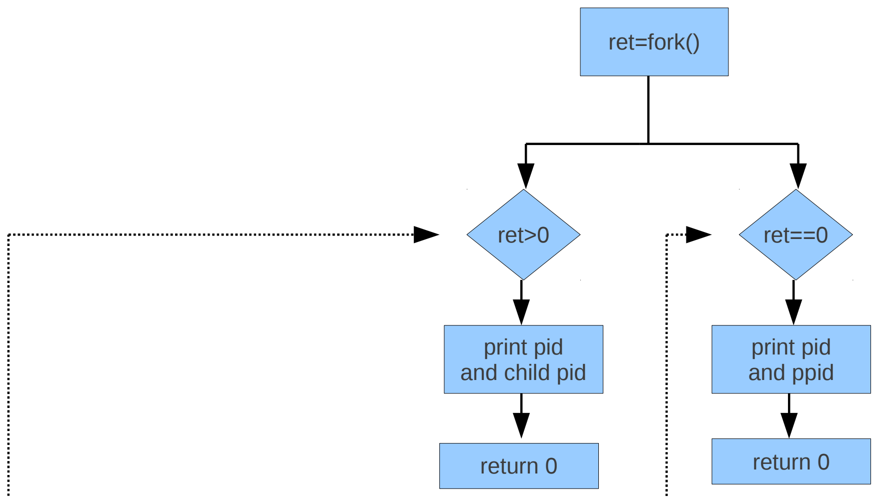
1. Concepts of process and process creation.
2. System calls for printing ids.

3.9.2 Objective

1. Understanding the picture of process creation.

3.9.3 Algorithm Design

1. Call fork. And collect the return value in ret.
2. If ret is 0, process is child. Else process is parent.
3. For both parent and child process, print thier pid and parent child ids.



```
int main()
{
        pid_t ret;
        ret = fork();

        if (ret == 0)
        {
                printf("child : pid = %d, ppid = %d\n", (int)getpid(), (int)getppid());
        }
        else if (ret != -1)
        {
                printf("Parent: pid = %d, child pid = %d\n", (int) getpid(), (int) ret);
        }
        return 0;
}
```

3.9.4 Dry Run

3.9.5 Practical Implementation

1. Process Creation

3.10 List of Assignments

(Id) / Date	Assignment Topic
( )	Do the man on ps and observe various options available
( )	Draw the family tree of the Process calling three forks()
( )	Create a process using System() system call
( )	Try all the functions from exec family
( )	Try all the functions from wait family
( )	Use the nice system call and analyze the result
( )	Create a Scenario which will cause a init to become a parent of the zombie process
( )	Try above example using waitid function
( )	Try to prevent the child from becoming the zombie without actually blocking on that
( )	Write a program to resume the process which has been stopped
( )	Create a child for the command(along with the options) passed as an argument from command line(eg:- ./a.out ls -l)
( )	Try to use the ptrace function in your program and use WUNTRACE option in the process parent process wait call
( )	Create three child processes and wait for all to terminate and print the status of each
( )	Write a C program to daemonise a process.
( )	WAP to demonstrate the usage of vfork()
( )	
( )	
( )	
( )	
( )	
( )	
( )	
( )	

Chapter 4

Signal

4.1 Signal

4.1.1 Signal Introduction / Concepts

Signalling is interrupt like mechanism. Its a way of sending simple messages to a process or group of processes. Most of these messages are already defined and can be found in <linux/signal.h>. However, signals can only be processed when the process is in user mode. If a signal has been sent to a process that is in kernel mode, it is dealt with immediately on returning to user mode.

There are a set of defined signals that the kernel can generate or that can be generated by other processes in the system, provided that they have the correct privileges. Each signal in Linux has got a name starting with 'SIG' and a number associated with it.

For example Signal 'SIGSEGV' has got a number 11.This is defined in /usr/include/bits/signum.h
You can list a system's set of signals using the kill command (kill -l).

An important characteristic of signals is that they may be sent at any time to a process whose state is usually unpredictable.Signals sent to a process that is not currently executing must be saved by the kernel until that process resumes execution. When switching from Kernel Mode to User Mode, Kernel will check whether a signal for a process arrived. This happens at almost every timer interrupt.
Notes:

4.1.2 Where Signals Come From

The kernel sends signals to processes in response to specific conditions. For instance, any of these Signals may be sent to a process that attempts to perform an illegal operation :
SIGBUS (bus error),
SIGSEGV (segmentation violation).
A Process may also send a Signal to another Process.
A Process may also send a Signal to itself.
Notes:

4.2 How Programs Responds to Signals

When a Process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.

For all possible signals, the system defines a default disposition or action to take when a signal occurs. There are four possible default dispositions:

- Exit: Forces the process to exit.
- Core: Forces the process to exit and create a core file.
- Stop: Stops the process.
- Ignore: Ignores the signal

Notes:

4.3 Handling Signals

4.3.1 Writing Your Own Signal Handler

There are several interfaces that allow you to register your own signal handler.

signal

```
signal(SIGALRM, wakeup);
alarm(10);
-------
-----doing some job -----
-------
```

While 10 seconds reached and process is still alive, wakeup function will get invoked.

sigaction

sigaction() is another system call that manipulates signal handler. It is much more advanced comparing signal().

```
struct sigaction action, oldact;
action.sa_sigaction = my_handler;
if (sigaction (SIGINT, &action, &oldact) == -1)
{
    perror ("sigaction fails");
    return -1;
}
```

Notes:

4.3.2 Signal in Signal Handler

Notes:

4.4 signal.h and Signal Related Functions

Notes:

4.4.1 raise()

sends a signal to the executing process.
E.g. raise(int sig);
Notes:

4.4.2 alarm()

Schedules an alarm signal. Can cause a SIGALRM.
E.g. alarm(int sec);
Notes:

4.4.3 pause()

Function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching or to terminate the process.
E.g. pause();
Notes:

4.5 Signal and Interrupts

Notes:

4.6 Process Termination and Exit Codes

Notes:

4.7 Practice - 1

Write a program to produce alarm. While alarm time expires, create an interface for reseting the alarm or for exiting from the program.

4.7.1 Prerequisite

1. signal system call concepts
2. alarm function concepts
3. time related functions concept

4.7.2 Objective

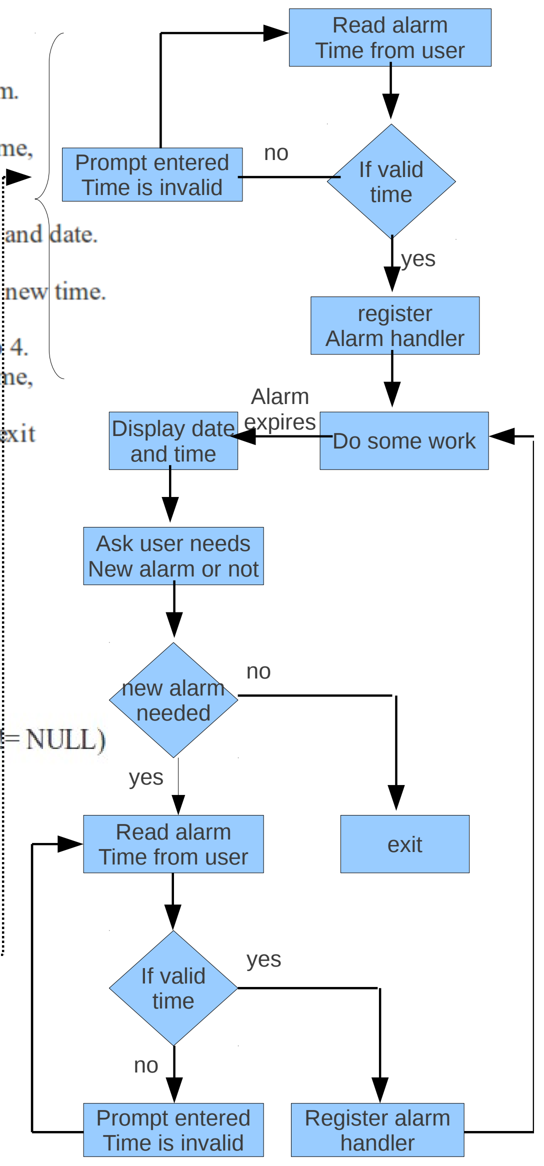
1. Implementing concepts of SIGALRM signal and time related functions.

4.7.3 Algorithm Design

1. Read alarm time from user.
2. Check alarm time is valid or not.
3. If so, register a signal handler for alarm.
4. Continue with some work.
5. If entered time is behind the current time, continue with step 1.

6. If alarm time expires, display the time and date.
7. Prompt for new alarm time.
8. If user need to set new alarm, read the new time.
9. Check the time is valid or not.
10. If yes set alarm and continue with step 4.
11. If entered time is behind the current time, continue with step 7.
12. If user had not entered any new time, exit from the program.

```
int main(int argc, char *argv[])
{
        unsigned int alarm_time;

        if (signal(SIGALRM, alarm_fun) != NULL)
        {
                perror("signal");
                return -1;
        }
        alarm_time = set_alarm();

        alarm(alarm_time);

        while (1);
        return 0;
}
```



```
int set_alarm()
{
        unsigned int hr, min, sec, alarm_time;
        struct tm *tm, tm_copy;
        time_t bak_time;

        while (1)
        {
                printf("Enter the alarm time (hr:min:sec): ");
                scanf("%u:%u:%u", &hr, &min, &sec);

                if (hr > 24 || min > 60 || sec > 60)
                {
                        printf("Entered time is Invalid\n");
                        continue;
                }

                bak_time = time(NULL);
                tm = localtime(&bak_time);
                tm_copy = *tm;

                tm_copy.tm_sec = sec;
                tm_copy.tm_min = min;
                tm_copy.tm_hour = hr;
                if (timelocal(&tm_copy) < bak_time )
                {
                        printf("Entered time is Invalid\n");
                        continue;
                }
                else
                        break;
        }

        alarm_time = timelocal(&tm_copy) - bak_time;
        return alarm_time;
}
```

```
void my_alarm(int signum)
{
        unsigned int alarm_time, choice;
        time_t time_var;

        time_var = time(NULL);
        printf("Alarm Reached %s\n", ctime(&time_var));

        printf("\n");
        printf("1. Set New Alarm :\n");
        printf("2. Exit\n");
        printf("Choice : ");
        scanf("%d", &choice);

        switch (choice)
        {
                case 1: alarm_time = set_alarm();
                        alarm(alarm_time);
                        break;
                case 2: exit(1);
        }
        return;
}
```

4.7.4 Dry Run

4.7.5 Practical Implementation

1. Scheduling task at certain intervals.

4.8 List of Assignments

(Id) / Date	Assignment Topic
( ) ______	Modify the Template program (SIGINT) to display the PID and uid of the process that sent a signal
( ) ______	WAP a program to handle the SIGSEGV and display the address which caused the segmentation fault
( ) ______	Try to handle the SIGABRT signal and have your observations
( ) ______	Kill the process by signal & try to get the signal number programatically
( ) ______	Handle the signal in process without using a sigaction function
( ) ______	Write a program to wait for the signal. Execution should begin only if signal is received
( ) ______	Write a program to handle SIGINT signal only once
( ) ______	Implement alarm program taking alarm time from command line
( ) ______	WAP to block certain signals from being received in signal handler
( ) ______	WAP to demonstrate the usage of clone
( ) ______	WAP to get the resource-usage information of the process & also of its child
( ) ______	WAP to block the parent process untill the child reaches the certain state of execution
( ) ______	WAP to find the default cancellation state of the thread
( ) ______	WAP which uses pause, alarm and raise functions
( ) ______	WAP to use the mapped memory
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	

Chapter 5

Threads

5.1 Introducation to Threads

5.1.1 What is Thread

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

As with processes, threads appear to run concurrently.
The Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute.
Notes:

5.1.2 Why Threads

- Threads, like processes, are a mechanism to allow a program to do more than one thing at a time.
- Conceptually, a thread exists within a process.
- Threads are a finer-grained unit of execution than processes.
- That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

This independent flow of control is accomplished because a thread maintains its own:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data

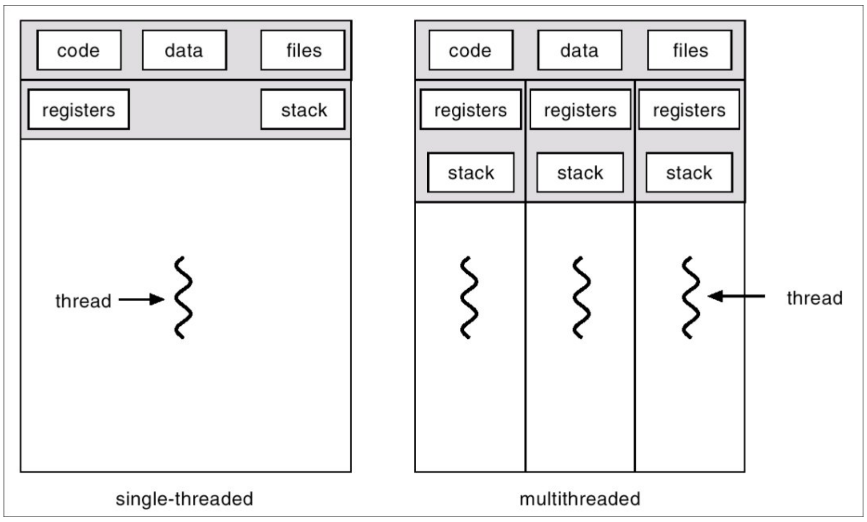
Notes:

5.1.3 Advantages of Threads

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways.
- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Asynchronous event handling.

Notes:

5.2 Multi Threaded Environment



Notes:

5.3 Process and Thread

- Less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Less communication overheads – communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads

Notes:

5.4 What is a pthread

GNU/Linux implements the POSIX standard thread API (known as pthreads).

All thread functions and data types are declared in the header file <pthread.h>. The pthread functions are not included in the standard C library. Instead, they are in libpthread, so you should add -lpthread to the command line when you link your program.

Notes:

5.5 When to Use Threading

Notes:

5.6 Thread Creation

Use the function pthread_create() to add a new thread of control to the current process. It is prototyped by:

int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void*(*start_routine)(void *), void *arg);

```
void *my_thread_func(void *arg)
{
int i = 0;
    while (i++ < 10000)
    {
    fprintf(stderr, "c");
    }
}

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, my_thread_func, NULL);

    while(1)
    ;
    return 0;
}
```

The pthread_create() function is called with attr having the necessary state behavior. start_routine is the function with which the new thread begins execution. When start_routine returns, the thread exits with the exit status set to the value returned by start_routine.

When pthread_create is successful, the ID of the thread created is stored in the location referred to as tid.
Notes:

5.7 Passing Data to Thread

Notes:

5.8 Returning Values From Threads

Notes:

5.9 Types of Threads

By default, created threads are joinable. That means, we can wait for its completion in any other thread using the function pthread_join():

This function shall suspend the calling thread until the targeted thread terminates. When value_ptr is non-NULL, then value_ptr shall contain the value returned by thread.

Following are some of the attributes that are specific to each thread :

- the thread ID, thread attribute, start routine, argument and return value
- thread scheduling policy and priority
- signal mask, alternate signal stack
- flags for cancellation, cleanup buffers
- keys for thread specific data
- errno

Notes:

5.10 Thread Attributes

Thread attributes are thread characteristics that affect the behavior of the thread.You can set the thread attributes at the time you start a thread or change them after the thread is actively running. Some common thread attributes and their effects are as follows:

- Priority
- Stack size
- Name
- Thread group
- Detach state
- Scheduling policy
- Inherit scheduling

Notes:

5.10.1 Creating Detached Threads

Pthreads offers a mechanism to tell the system: I am starting this thread, but I am not interested about joining it. Please perform any clean-up action for me, once the thread has terminated. This operation is called detaching a thread. We can detach a thread as follows:

- During thread creation using the detachstate thread attribute
- From any thread, using pthread_detach()

The function pthread_detach() can be called from any thread, in particular from within the thread to detach (any thread can obtain its own thread ID using the pthread_self() API).

To create a thread in detach state, we set the detachstate thread attribute with the function pthread_attr_setdetachstate() to PTHREAD_CREATE_DETACHED. This is shown below:

```
pthread_attr_t attr;
int main()
{
    pthread_t tid;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&tid, &attr, my_thread_func, NULL);

    pthread_join(tid, NULL);
}
```

The most important rules for join/detach are:

- Dont join a thread that has been already joined
- Dont join a detached thread
- If you detach a thread, you cannot re-attach it

<u>Notes:</u>

5.11 Self Thread ID

The function pthread_self() can be called to return the ID of the calling thread.

```
pthread_t tid;
tid = pthread_self();
```

Notes:

5.12 Thread Cancellation

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see pthread_create()
- By calling pthread_exit(), supplying an exit status
- By termination with POSIX cancel functions; see pthread_cancel()

The void pthread_exit(void *status) is used to terminate a thread in a similar fashion the exit() for a process.

```
int status;
pthread_exit(&status); /* exit with status */
```

The pthread_exit() function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by status are retained until the thread is waited for (blocked). Otherwise, status is ignored and the thread's ID can be reclaimed immediately.
Notes:

5.12.1 How to Cancel

int pthread_cancel(pthread_t thread);

```
pthread_t thread;
int ret;
ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions,
pthread_setcancelstate() and pthread_setcanceltype() determine that state.
Notes:

5.12.2 Consequence of Thread Cancellation

Notes:

5.12.3 Controlling Cancellation

Notes:

5.12.4 Implementing Critical Section

Notes:

5.13 Cleanup Handlers

Cleanup handlers are functions that get called when a thread terminates, either by calling pthread_exit or because of cancellation.The purpose of cleanup handlers is to free the resources that a thread may hold at the time it terminates.

```
void *my_thread_func(void *arg)
{
    pthread_cleanup_push(my_cleanup, NULL);
    --------
    -----do some work ---
    --------
    pthread_cleanup_pop(0);
}
```

Notes:

5.14 List of Assignments

(Id) / Date	Assignment Topic
( ) ______	Write a code to pass a number to a thread, computer the factorial of the number and return the result to the main thread
( ) ______	Create a detached thread and pthread_join on that and observe the effect
( ) ______	Write a program to prevent the critical section from being canceled
( ) ______	Show through a code the importance of cleanup handlers
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	

Chapter 6

Synchronization

6.1 What is Synchronization

Programming with threads is very tricky because most threaded programs are concurrent programs. In particular, there is no way to know when the system will schedule one thread to run and when it will run another.

One thread might run for a very long time, or the system might switch among threads very quickly.

Debugging a threaded program is difficult because you cannot always easily reproduce the behavior that caused the problem. You might run the program once and have everything work fine; the next time you run it, it might crash.

There is no way to make the system schedule the threads exactly the same way it did before.
Notes:

6.2 Why Synchronization

6.2.1 Race Condition

The ultimate cause of most bugs involving threads is that the threads are accessing the same data. So the powerful aspects of threads can become a danger.

If one thread is only partway through updating a data structure when another thread accesses the same data structure, its a problem.

These bugs are called race conditions; the threads are racing one another to change the same data structure.
Notes:

6.2.2 Critical Section

A critical section is a segment of code in which a thread may be updating or reading data that other threads are dependent on, but one thread at a time. Thus we can prevent the race condition problem.

The characteristic properties of the code that form a Critical Section are

- Codes that reference one or more variables in a read-update-write fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in read-updata-write fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it is possibly in use by another thread.

Notes:

6.3 Mutexes

6.3.1 What is Mutex

The solution to the race condition problem is to let only one thread access the resource at a time.

GNU/Linux provides mutexes, short for MUTual EXclusion locks.

A mutex is a special lock that only one thread may lock at a time.

If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is blocked, or put on hold.

Only when the first thread unlocks the mutex is the second thread unblockedallowed to resume execution.
Notes:

6.3.2 Creating / Destroying Mutexes

To create a mutex, create a variable of type pthread_mutex_t and pass a pointer to it to pthread_mutex_init.

The second argument to pthread_mutex_init is a pointer to a mutex attribute object, which specifies attributes of the mutex.

```
pthread_mutex_t mut;
int main()
{
    pthread_mutex_init(&mut, NULL);
    pthread_create(*tid, NULL, my_thread_func, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

Notes:

6.3.3 Locking / Unlocking Mutexes

A thread may attempt to lock a mutex by calling pthread_mutex_lock on it.

- If the mutex was unlocked, it becomes locked and the function returns immediately.
- If the mutex was locked by another thread, pthread_mutex_lock blocks execution and returns only eventually when the mutex is unlocked by the other thread.
- More than one thread may be blocked on a locked mutex at one time.
- When the mutex is unlocked, only one of the blocked threads is unblocked and allowed to lock the mutex; the other threads stay blocked.

A call to pthread_mutex_unlock unlocks a mutex.

This function should always be called from the same thread that locked the mutex.

```
void *my_thread_func(void *arg)
{
    pthread_mutex_lock(&mut);
    ----------
    -----do some work ---
    ----------
    pthread_mutex_unlock(&mut);
}
```

Notes:

6.3.4 Deadlocks and Starvation

Mutexes provide a mechanism for allowing one thread to block the execution of another.This opens up the possibility of a new class of bugs, called deadlocks. A deadlock occurs when one or more threads are stuck waiting for something that never will occur.

Starvation occurs when a scheduler process (i.e. the operating system) refuses to give a particular thread any quantity of a particular resource (generally CPU). If there are too many high-priority threads, a lower priority thread may be starved. This can have negative impacts, though, particularly when the lower-priority thread has a lock on some resource.

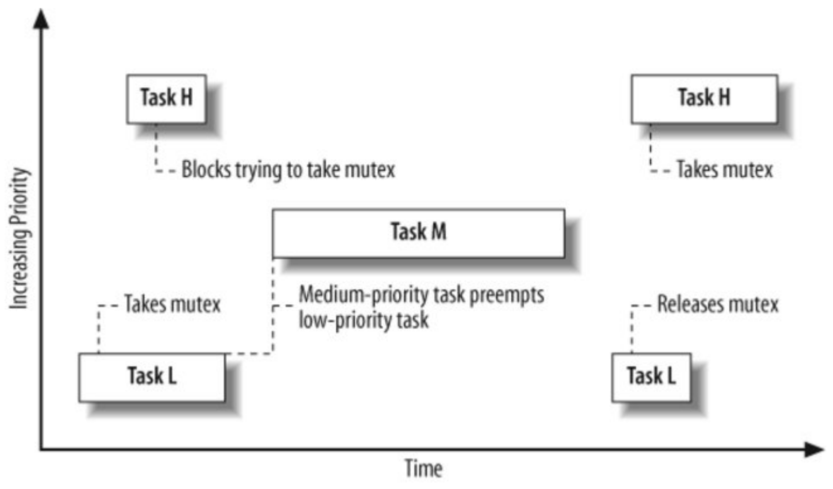
Notes:

6.3.5 Priority Inversion

Priority inversion is a problematic scenario in scheduling when a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks.

Consider there is a task L, with low priority. This task requires resource R. Consider that L is running and it acquires resource R. Now, there is another task H, with high priority. This task also requires resource R. Consider H starts after L has acquired resource R. Now H has to wait until L relinquishes resource R. Everything works as expected up to this point, but problems arise when a new task M starts with medium priority during this time.

At this stage, H is blocked on R, M is ready to run, L has acquired R. Since M is highest priority unblocked task currently, it will be scheduled first and it will eat up all the processing power until it finishes, not allowing any other task to run. This would block L from running. Since L cannot run, L cannot relinquish R. Since R is still in use (by L), H cannot run. So as you see above, M will run till it is finished, then L will run - at least up to a point where it can relinquish R - and then H will run. Thus, in above scenario, tasks with lower priority run before task with high priority, effectively giving us a priority inversion.



Notes:

6.3.6 Types of Mutexes and Mutex Attributes

Fast mutex (the default kind) will cause a deadlock to occur.

Recursive mutex - safely be locked many times by the same thread. The mutex remembers how many times pthread_mutex_lock was called on it by the thread that holds the lock; that thread must make the same number of calls to pthread_mutex_unlock before the mutex is actually unlocked and another thread is allowed to lock it.

Error-checking mutex - The second consecutive call to pthread_mutex_lock returns the failure code EDEADLK.
Notes:

6.3.7 Non-Blocking Mutex Tests

GNU/Linux provides pthread_mutex_trylock for this purpose. If you call pthread_mutex_trylock on an unlocked mutex, you will lock the mutex as if you had called pthread_mutex_lock, and pthread_mutex_trylock will return zero.

If the mutex is already locked by another thread, pthread_mutex_trylock will not block. Instead, it will return immediately with the error code EBUSY. The mutex lock held by the other thread is not affected. You may try again later to lock the mutex.
Notes:

6.4 Semaphores for Threads

A semaphore is a counter that can be used to synchronize multiple threads. As with a mutex, GNU/Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.

Each semaphore has a counter value, which is a non-negative integer given at initialising.

sem_init(sem_t *sem, int pshared, unsigned int value);

A wait operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphores value becomes positive, it is decremented by 1 and the wait operation returns.

sem_wait(sem_t *sem);

A post operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphores value back to zero).

sem_post(sem_t *sem);

Notes:

6.5 Practice - 1

Write a multi threaded program to calculate the average of numbers in an array.

6.5.1 Prerequisites

1. Thread creation concepts.
2. Thread synchronization methods.

6.5.2 Objective

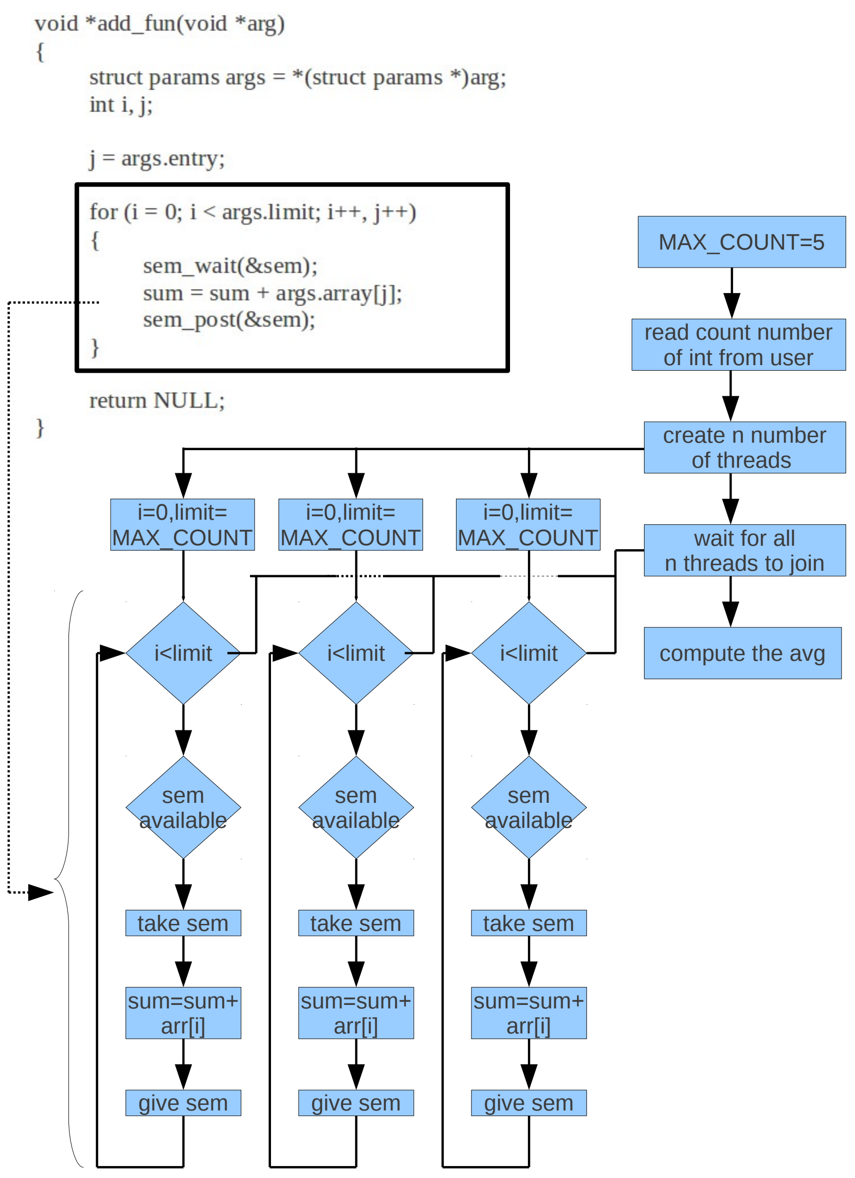
1. Implementing synchronization between threads.

6.5.3 Algorithm Design

1. Read the count of integers from command line.
2. Allocate an array of count length.
3. Decide the number of integers needed to handle by a single thread by using a macro, MAX_COUNT_THREAD.
4. Declare a global variable named sum.
5. Read the count number of integers from user and store them into array.
6. Create n number of threads, such that per thread can handle maximum 4 number of integers.
7. Pass starting point in the array, number of elements to handle in the array, and the array itself as the arguments to the thread function.
8. In each thread calculate sum of MAX_COUNT_THREAD integers and update them in shared variable sum.
9. Use synchronization methods while accessing shared variable sum.

10. After termination of all threads, calculate average of sum.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_COUNT_THREAD 5

struct params
{
        unsigned int entry;
        unsigned int limit;
        int *array;
};
int sum;
sem_t sem;
```



```
void *add_fun(void *arg)
{
        struct params args = *(struct params *)arg;
        int i, j;

        j = args.entry;
        for (i = 0; i < args.limit; i++, j++)
        {
                sem_wait(&sem);
                sum = sum + args.array[j];
                sem_post(&sem);
        }

        return NULL;
}
```

```
int main(int argc, char *argv[])
{
        int count, i;

        if (argc < 2)
        {
                printf("Usage : ./a.out <count of integers>\n");
                return 1;
        }

        count = atoi(argv[1]);

        int arr[count];
        pthread_t tid[count  /  MAX_COUNT + 1];
        struct params arg[count / MAX_COUNT + 1];
        double avg = 0.0;

        sem_init(&sem, 0, 1);

        for (i = 0; i < count; i++)
        {
                scanf("%d",  &arr[i]);
        }

        for (i = 0; i < (count / MAX_COUNT) ; i++)
        {
                arg[i].entry = i * MAX_COUNT;
                arg[i].limit = (count – (i * MAX_COUNT)  >=  MAX_COUNT) ?
                        MAX_COUNT: count – (i * MAX_COUNT);
                arg[i].array = arr;
                pthread_create(&tid[i], NULL, add_fun, &arg[i]);
        }

        for (i = 0 ; i < (count / MAX_COUNT) + 1 ;  i++)
        {
                pthread_join(tid[i], NULL);
        }

         printf("%d\n", sum);
         avg = (double)sum / count;
         printf("Average of the entered numbers are %f\n", avg);

         return 0;
}
```

6.5.4 Dry Run

6.5.5 Practical Implementation

1. Multi threaded programming.

6.6 List of Assignments

(Id) / Date	**Assignment Topic**
( ) ____	Create a deadlock with two threads and two mutex
( ) ____	Use semaphore to synchronize the critical section access between two threads
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	
( ) ____	

Chapter 7

LINUX IPC

7.1 Introduction

Interprocess communication (IPC) is the transfer of data among processes. The types of inter process communication are:

- Signals - Sent by other processes or the kernel to a specific process to indicate various conditions.
- Pipes - Unnamed pipes set up by the shell normally with the "|" character to route output from one program to the input of another.
- FIFOS - Named pipes operating on the basis of first data in, first data out.
- Message queues - Message queues are a mechanism set up to allow one or more processes to write messages that can be read by one or more other processes.
- Semaphores - Counters that are used to control access to shared resources. These counters are used as a locking mechanism to prevent more than one process from using the resource at a time.
- Shared memory - The mapping of a memory area to be shared by multiple processes.

Notes:

7.2 Pipe

Pipe is used for one-way communication of a stream of bytes.

7.2.1 Creating Pipes

To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline.

int pipe(int fd[2]);
The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

To send data to the pipe, we use the write() system call, and to retrieve data from the pipe, we use the read() system call. Remember, low-level file I/O system calls work with file descriptors! However, keep in mind that certain system calls, such as lseek(), do not work with descriptors to pipes.

```
int pfd[2];
pipe(pfd);

write(pfd[1], buf, size);
read(pfd[0], buf, SIZE);
```

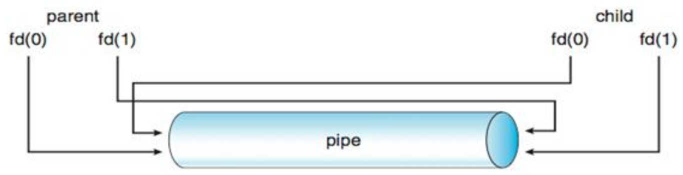
Pipe can be used to connect the standard output of one process to the standard input of another.They provide a method of one-way communications (hence the term half-duplex) between processes.

ls | sort | lp
The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp.
Notes:

7.2.2 Properties

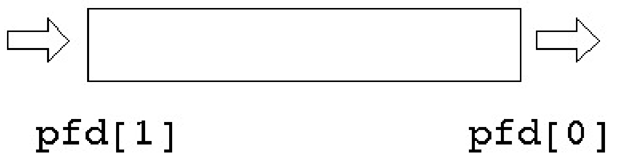
Pipe does a destructive read, which means that the data once read from the pipe cannot be retrieved. A pipe has a finite size, always atleast 4K.
Notes:
Notes:

7.2.3 Process Communication Using Pipes

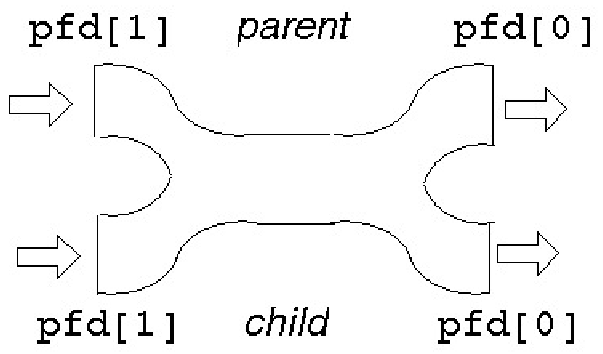


A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using fork(). A pipe opened before the fork becomes shared between the two processes.

Before fork



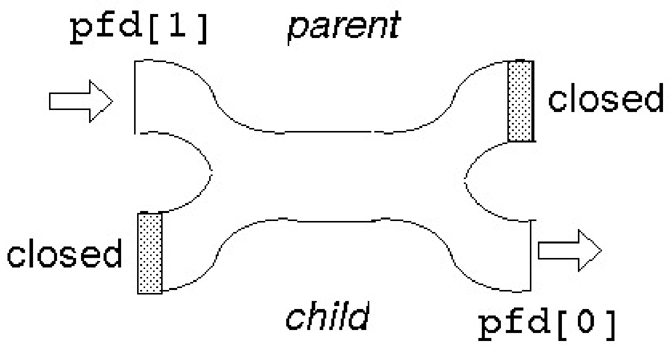
After fork



This gives two read ends and two write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.

Either process can write into the pipe, and either can read from it. Which process will get what is not known.

For predictable behaviour, one of the processes must close its read end, and the other must close its write end. After fork



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.

```
#include <stdio.h>

#define SIZE 1024

int main(int argc, char **argv)
{
    int pfd[2];
    int nread;
    int pid;
    char buf[SIZE];

if (pipe(pfd) == -1)
{
    perror("pipe failed");
    exit(1);
}
if ((pid = fork()) < 0)
{
    perror("fork failed");
    exit(2);
}

if (pid == 0)
{
        /* child */
        close(pfd[1]);
        while ((nread = read(pfd[0], buf, SIZE)) != 0)
        printf("child read %s\n", buf);
        close(pfd[0]);
    }
    else
    {
        /* parent */
        close(pfd[0]);
        strcpy(buf, "hello...");
        /* include null terminator in write */
        write(pfd[1], buf,
        strlen(buf)+1);
        close(pfd[1]);
    }
        exit(0);
    }
```

Notes:

7.2.4 Limitations of Pipes

Pipes can be used only between the inter-related processes, like parent and child.
Notes:

7.3 FIFOs

Fifos are named pipes.
Notes:

7.3.1 Properties

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

Notes:

7.3.2 Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

To create a FIFO in C, we can make use of the mknod() system call.

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

Notes:

7.4 Shared Memory

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.
Notes:

7.4.1 Allocation

shmget() is used to obtain access to a shared memory segment. It is prototyped by:
int shmget(key_t key, size_t size, int shmflg);

The key argument is a access value associated with the semaphore ID. The size argument is the size in bytes of the requested shared memory. The shmflg argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

```
int shm_id;
shm_id = shmget(key, 1024, IPC_CREAT);
```

Notes:

7.4.2 Attachment and Detachment

shmat() and shmdt() are used to attach and detach shared memory segments.

Their prototypes are as follows:
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr.

```
void *ptr;
ptr = shmat(shm_id, 0, 0);
-------
---do some work ---
-------
shmdt(ptr);
```

Notes:

7.4.3 Controlling and Deallocating Shared Memory

shmctl() is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:
int shmctl(int shmid, int cmd, struct shmid_ds *buf);

```
struct shmid_ds shm;
shmctl(shm_id, IPC_STAT, &shm);
--------
--- do some work ----
--------
shmdt(ptr);
shmctl(shm_id, IPC_RMID, &shm);
```

Notes:

7.5 Semaphores

Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it.
Notes:

7.5.1 Allocation

- The calls semget and semctl allocate and deallocate semaphores
- Invoke semget with a key specifying a semaphore set, the number of semaphores in the set, and permission flags as for shmget
- The return value is a semaphore set identifier.
- You can obtain the identifier of an existing semaphore set by specifying the right key value; in this case, the number of semaphores can be zero.

```
int main()
{
    int semid;
    semid = semget(key, 2, IPC_CREAT|0600);
}
```

Notes:

7.5.2 Deallocation

- Semaphores continue to exist even after all processes using them have terminated.
- Invoke semctl with the semaphore identifier, the number of semaphores in the set, IPC_RMID as the third argument, and any union semun value as the fourth argument (which is ignored).

```
int main()
{
    /* Allocate and get semid */

    semctl(semid, 0, IPC_RMID);
}
```

Notes:

7.5.3 Initialization

- To initialize a semaphore, use semctl with zero as the second argument and SETALL as the third argument.
- For the fourth argument, you must create a union semun object and point its array field at an array of unsigned short values.
- Each value is used to initialize one semaphore in the set.

```
int main()
{
    /* Allocate */

    unsigned short int arr[2];
    union semun sem;
    arr[0] = 1;
    arr[1] = 1;
    sem.array = arr;
    semctl (semid, 1, SETALL, sem.array);

    /* Deallocate */
}
```

Notes:

7.5.4 Wait and Post operations

- Each semaphore has a non-negative value and supports wait and post operations.
- The semop system call implements both operations.
- Its first parameter specifies a semaphore set identifier.
- Its second parameter is an array of struct sembuf elements, which specify the operations you want to perform.
- The third parameter is the length of this array.
- The fields of struct sembuf are listed here:
  - sem_num is the semaphore number in the semaphore set on which the operation is performed.
  - sem_op is an integer that specifies the semaphore operation.
  - If sem_op is a positive number, that number is added to the semaphore value immediately.
  - If sem_op is a negative number, the absolute value of that number is subtracted from the semaphore value.

```
int main()
{
    /* Allocate */
    /* Initialise */

    struct sembuf buffer[1];
    buffer[0].sem_num = 0;
    buffer[0].sem_op = -1;
    buffer[0].sem_flg = SEM_UNDO;
    semop(semid, &buffer[0], 1);

    /* Deallocate */
}
```

Notes:

7.6 List of Assignments

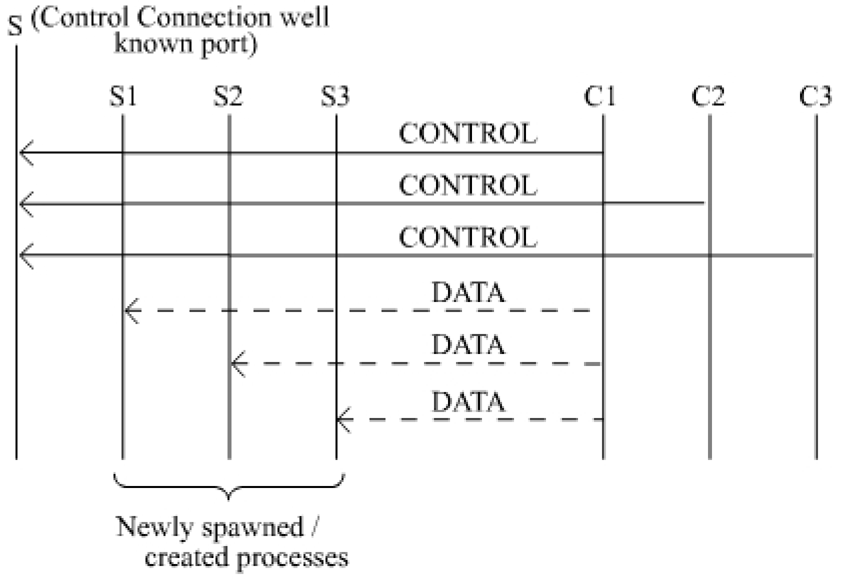
(Id) / Date	**Assignment Topic**
( ) ______	WAP to have a IPC using Message queues
( ) ______	Implement ls -l | grep "pattern" | wc -l
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	
( ) ______	

Chapter 8

Unix network programming

8.1 Introduction to sockets

8.2 Client server mechanism:


S (Control Connection well known port)
S1
S2
S3
C1
C2
C3
CONTROL
CONTROL
CONTROL
DATA
DATA
DATA
Newly spawned / created processes

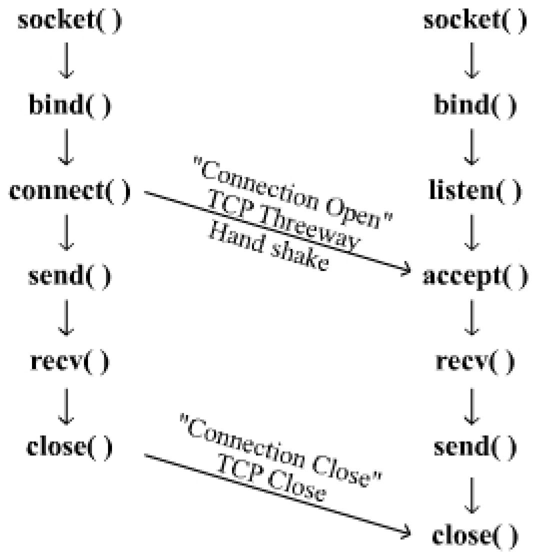
8.3 Structures used in socket programming:

```
struct sockaddr
{
        unsigned short    sa_family;    // address family, AF_xxx
        char              sa_data[14];  // 14 bytes of protocol address
};

struct sockaddr_in
{
        short            sin_family;   // e.g. AF_INET
        unsigned short   sin_port;     // e.g. htons(3490)
        struct in_addr   sin_addr;     // see struct in_addr, below
        char             sin_zero[8];  // zero this if you want to
};

struct in_addr
{
        unsigned long s_addr; // that's a 32-bit long, or 4 bytes
};
```

Hands on:

1. Open sys/types.h and sys/socket.h in the Linux and read through the definitions
2. What is the byte ordering that Linux is using? Write a C program to find out the byte ordering of Linux
3. What is the IP address of the loopback interface ?

8.4 The socket APIs:



- int socket(int domain, int type, int protocol);
- int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
- int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
- int listen(int sockfd, int backlog);
- int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
- int send(int sockfd, const void *msg, int len, int flags);
- int recv(int sockfd, void *buf, int len, unsigned int flags);
- int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklen_t tolen);
- int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
- close(sockfd);

8.5 Trivial functions:

- int gethostname(char *hostname, size_t size);
- struct hostent *gethostbyname(const char *name);
- uint32_t htonl(uint32_t hostlong);
- uint16_t htons(uint16_t hostshort);
- uint32_t ntohl(uint32_t netlong);
- uint16_t ntohs(uint16_t netshort);
- int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
- int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);

8.6 Concurrent & Iterative Servers

8.7 I/O multiplexing

8.7.1 The select() System call

```
int select(int numfds, fd_set *readfds, fd_set *writefds,
                       fd_set *exceptfds, struct timeval *timeout);
```

```
struct timeval
{
        int tv_sec;     // seconds
        int tv_usec;    // microseconds
};
```

- FD_ZERO(fd_set *set) – clears a file descriptor set
- FD_SET(int fd, fd_set *set) – adds fd to the set
- FD_CLR(int fd, fd_set *set) – removes fd from the set
- FD_ISSET(int fd, fd_set *set) – tests to see if fd is in the set

8.7.2 Examples for select()

```
/*
** select.c -- a select() demo
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input

int main(void)
{
        struct timeval tv;
        fd_set readfds;

        tv.tv_sec = 2;
        tv.tv_usec = 500000;

        FD_ZERO(&readfds);
        FD_SET(STDIN, &readfds);

        // don't care about writefds and exceptfds:
        select(STDIN+1, &readfds, NULL, NULL, &tv);

        if (FD_ISSET(STDIN, &readfds))
                printf("A key was pressed!\n");
        else
                printf("Timed out.\n");

        return 0;
}
```

8.8 Daemons in Linux operating system and writing a daemon:

Xinetd Xinetd is a secure replacement for inetd, the Internet services daemon. Xinetd provides access control for all services based on the address of the remote host and/or on time of access and can prevent denial-of-access attacks. Xinetd provides extensive logging, has no limit on the number of server arguments, and lets you bind specific services to specific IP addresses on your host machine. Each service has its own specific configuration file for Xinetd; the files a relocated in the /etc/xinetd.d directory

8.9 The Internet superserver xinetd:

The xinetd daemon is a TCP wrapped super service which controls access to a subset of popular network services including FTP, IMAP, and Telnet. It also provides service-specific configuration options for access control, enhanced logging, binding, redirection, and resource utilization control. When a client host attempts to connect to a network service controlled by xinetd, the super service receives the request and checks for any TCP wrappers access control rules. If access is allowed, xinetd verifies that the connection is allowed under its own access rules for that service and that the service is not consuming more than its allotted amount of resources or is in breach of any defined rules. It then starts an instance of the requested service and passes control of the connection to it. Once the connection is established, xinetd does not interfere further with communication between the client host and the server.

8.9.1 The /etc/services

This configuration file defines the sockets and protocols used for Internet services. Each service is listed on a single line corresponding to the form:

Syntax: ServiceName PortNumber/ProtocolName Aliases

ServiceName : Specifies an official Internet service
PortNumber : Specifies the socket port number user
ProtocolName : Specifies the transport protocol used
Aliases : Specifies a list of unofficial service names.

8.9.2 The /etc/xinetd.conf

The /etc/xinetd.conf file contains general configuration settings which effect every service under xinetd's control. It is read once when the xinetd service is started, so for configuration changes to take effect, the administrator must restart the xinetd service. Below is a sample /etc/xinetd.conf

```
service <service_name>
{
        <attribute> <assign_op> <value> <value>
}

service SMTP
{
        socket type   = stream
        protocol =  tcp
        wait =  no
        user =  mail
        server =  /usr/sbin/exim
}
```

8.10 List of Assignments

(Id) / Date	Assignment Topic
( ) ___	Simple client-server using TCP
( ) ___	Simple client-server using UDP
( ) ___	Concurrent TCP server using fork()
( ) ___	Terminal protocol using sockets
( ) ___	Remote command execution using UDP sockets
( ) ___	Simple program using select()
( ) ___	Concurrent TCP server using select()
( ) ___	UDP echo client using select()
( ) ___	Implementation of a daemon using 'xinetd'

4. Terminal protocol using sockets

Assignment description: The idea of terminal protocol is to implement a protocol using sockets. In order to implement following three defnitions are made:

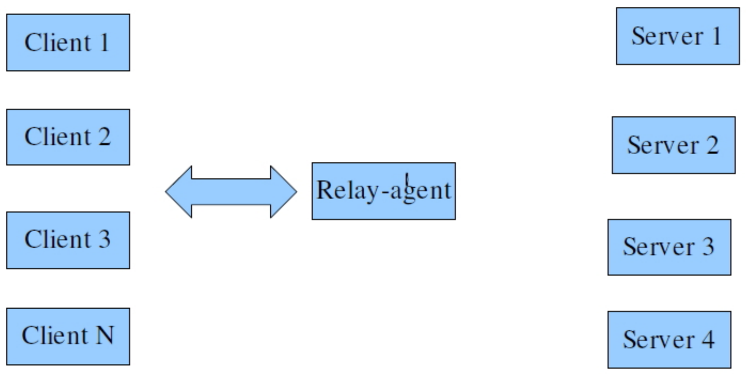
Servers:

These are the server entities who are running a particular service in a predefined port. The service could be anything and the student need to decide it. Say for example it can provide quote of the day, time of the day or simple database query. While implementing the service usage of data structures library needs to be made. No array based or any static storage is allowed in servers. All servers are designed as concurrent servers.

Clients:

The clients are the ones who need any of the services provided by the server. But unlike the normal client-server case, these clients don't know the port number and the IP address of the server. They only know the intermediate 'relay-agent' (which is again a typical socket server program) running on a pre-defned port. The clients would send a 'relay request' packet to the relay agent along with the service name and number. These service name and number definitions are user defined. On getting the 'relay reply' packet, the

client should parse it and find out the port number and IP address of the server. Depending on the service it requires it sends the appropriate 'server request' packets and get the 'server reply' packets back.
Relay-agent: The relay agents act in between the clients and servers. They have a database of server information and would provide to the clients, when clients send a 'relay request' packet. The relay agent would search through its database for the required service and send back the response in the 'relay reply' packet. The database is again a dynamically created and data structures library must be used for the same.



Implementation details:

1. There should be a client program (client.c), relay program (relay.c) and server programs (server1.c, server2.c etc..) as a part of the deliverables.
2. The implementation should use TCP sockets in all communications.
3. There should be relay request, relay reply, service request(s) and service reply(s) packets should be defined.
4. Under no circumstances any static storage should be used.
5. Assume the connectivity is robust and no need to handle the error cases.

Remote command execution using UDP sockets

Assignment description:

In the network diagnostics kind of applications, monitoring remote devices and getting by executing certain commands play a very important role. For example, sitting in India the administrator may want to monitor the enterprise server that is kept in Russia. The adminis- trator would send the remote command and the external entity would reply back with the output of the command. This assignment is to implement the remote command execution using Linux environment.

The UDP client would send any standard Linux shell command (say 'ls') using the 'command request' packet along with the number of times the command needs to be executed. At the server side the server would parse the command the execute it the number of times specified. Apart from ex- ecuting it, the server would capture the output in a file. This file would be then read by the server and the output would be sent back to the client in terms of 64 bytes of 'data' packets along with the packet number. If the data size is less then 64 bytes it needs to be specified by the server. As this connection is unreliable UDP connection the client should acknowledge back with and ACK packet along with the packet number. The server would send the next packet only after receiving the successful ACK for the previous packet.

Implementation details:

1. Separate command request, data and ACK packets needs to be defined.
2. Output of each command should be written in a Separate file both in the client side and server side.
3. Use C file operations

8.UDP echo client using 'select'

Assignment description: Construct a UDP echo client that takes as ar

guments a server name or IP address and a port number.Your client will use select() to determine if data is ready to be read from the keyboard.Any line typed on the keyboard should be sent to the server (using sendto() API). Any datagrams received from the server (recvfrom()) should be printed (dis played). Maximum message/buffer length is 1024 bytes. For example:

```
$echo client 192.168.32.10 1000
message to the server
> ............
> ............
> message from the server
```

Chapter 9

Process Management

9.1 Scheduling Introduction

Schedular is the component of an Operating System that determines which process should be run, and when. It is a mechanism used to achieve the desired objective of multitasking. So schedular is termed as the heart and soul of the Operating System.
Notes:

9.2 Basic Types of Schedulers

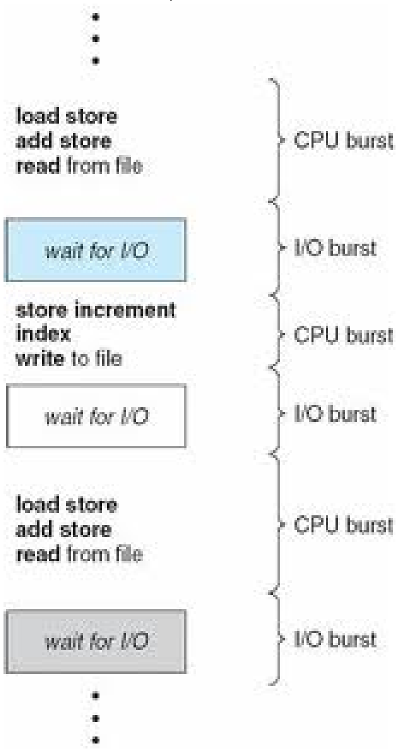
- Long-term scheduler (or job scheduler) selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) selects which process should be executed next and allocates CPU.

Notes:

9.3 Process Execution Cycle

Maximum CPU utilization obtained with multiprogramming.

CPU & I/O Burst Cycle Process execution consists of a cycle of CPU execution and I/O wait.



Notes:

9.4 When Schedular is called

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. CPU scheduling decisions may take place when a process:

- 1.Switches from running to waiting state.
- 2.Switches from running to ready state.
- 3.Switches from waiting to ready.
- 4.Terminates.

Scheduling under 1 and 4 is nonpreemptive.All other scheduling is preemptive.
Notes:

9.5 Scheduling Types

- Co-Operative
- Pre-Emptive
- First Come First Serve
- Priority based
- RR-Time slice
- RR-Priority based
- Pre-emptive

Notes:

9.6 Scheduling Algos

9.6.1 Co-operative versus Preemptive

In Co-operative scheduling, Process co-operate in terms of sharing processor timing. The process voluntarily gives the kernel a chance to perform a process switch.

In Preemptive Multiprocessing, process are preempted by a portion of the OS called the scheduler.
Notes:

9.6.2 Draw-backs of co-operative scheduling

- Failure to make switch call affects all other tasks in the system.
- More importantly, if a task fails (crashes), the call to the scheduler is never made and the system locks up.
- The only advantage of co-operative scheduling is fewer reentrance problems are encountered.

Notes:

9.6.3 First Come First Serve

FCFS is also known as FIFO.In FCFS, all the process have equal priorities and they share the processor co-operatively.The process are scheduled for execution in the order which they enter the ready queue.
E.g : If the process enter the ready queue in the order P1, P2 & P3 then the order of execution would be P1, P2 and P3.

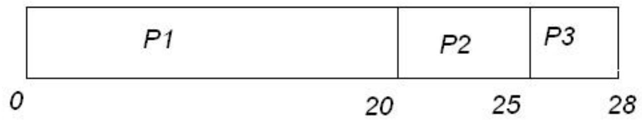
- The average waiting time under the FCFS policy is often quite long.
- FCFS is non-preemptive.

example :

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	**Burst time**
P1	20
P1	5
P2	3

Notes:



Notes:

9.6.4 Round Robin Time Sliced

This time slicing is similar to FCFS except that the scheduler forces process to give up the processor based on the timer interrupt. It does so by preempting the current process (i.e. the process actually running) at the end of each time slice.The process is moved to the end of the priority level.
Notes:

9.6.5 Round Robin Priority Based

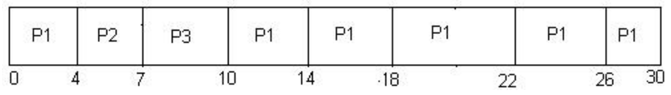
Process t1, t2, & t3 having same priorities. Round-robin scheduling. Process t4 has higher priority. So it preempts t2.

example :

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst time
P1	24
P2	3
P3	3

Notes:



Notes:

9.6.6 Pre-emption

Pre-emption means while a lower priority process is executing on the processor another process higher in priority than comes up in the ready queue, it preempts the lower priority process.
Notes:

9.6.7 Priority Based Non-Preemptive

In Priority based non-preemptive scheduling, while a lower priority process is executing even if a higher priority process comes up in the ready queue it wont preempt the current process but will give control to it either if it finishes its job or has to wait for some I/O.

If there are five process in a systems with increasing order of priority as t1,t2,t3,t4 and t5 where t5 is the highest priority.At one point of time t3 is executing and the t2 & t1 are in the ready queue where as t4 and t5 are in the delayed state. As the delay for t5 is over it comes in to the ready queue but does not acquire the processor from t3 until t3 finishes or goes into I/O wait/delayed state.
Notes:

9.6.8 Priority Based Preemptive

In case priority based preemptive algorithm if a higher priority process becomes ready while a lower priority process is executing it immediately preempts the lower priority process from the processor. It is widely used method for embedded systems s/w.
Notes:

9.7 Advanced Scheduling

9.7.1 Rate Monotonic Scheduling

The highest Priority is assigned to the Task with the Shortest Period. All Tasks in the task set are periodic.The relative deadline of the task is equal to the period of the Task. Smaller the period, higher the priority i.e Priority is inversely proportional to the period.
Notes:

9.7.2 Earliest Deadline First Scheduling

This kind of scheduler tries to give execution time to the task that is most quickly approaching its deadline. This is typically done by the scheduler changing priorities of tasks on-the-fly as they approach their individual deadlines.
Notes:

9.8 Linux Scheduling Algo

Works by dividing the CPU time into epochs. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. Different processes have different time quantum durations.

The time quantum value is the maximum CPU time portion assigned to the process in that epoch.When a process has exhausted its time quantum, it is preempted and replaced by another runnable process.If it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch.

The epoch ends when all runnable processes have exhausted their quanta;Then the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.
Notes:

Chapter 10

Memory Management

10.1 Introduction

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.
Notes:

10.2 Memory Management Requirements

- Relocation
  - Programmer does not know where the program will be placed in memory when it is executed
  - Before the program is loaded, address references are usually relative addresses to the entry point of the program, which is 0. These are called logical addresses, which make up a logical address space.
  - Memory references must be translated in the code to actual physical memory addresses, which make up a physical address space.

  The Translation may be performed at:

  - Compile Time: If it is known in advance that the program will reside at specific location in main memory, then compiler may be told to build the object code with absolute addresses.
  - Load Time: In most cases, the compiler must generate relocatable code with logical addresses. The address translation may be performed at the load time.
  - Execution Time: The process may be swapped out to allow other processes to be loaded.
- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Impossible to check absolute addresses in programs since the program could be relocated
  - Must be checked during execution
- Sharing
  - Allow several processes to access the same portion of memory.
  - For example, when using shared memory IPC, we need two processes to share the same memory segment
- Logical Organization(Memory with attributes)
  - Program is divided into modules, each having a different attribute.
  - For example, in Linux, Code Segment has a Read-only attribute.

- Physical Organization(Overlaying & Reusing)
  - Processes in the user space will be leaving & getting in.
  - Each process needs the memory to execute. So, the memory needs to be partitioned between processes.

Notes:

10.3 Memory Management Techniques

10.3.1 Fixed Partioning

Notes:

10.3.2 Dynamic Partioning

Notes:

10.4 Virtual Addressing

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called pages.
Notes:

10.4.1 Virtual Vs Physical Addressing

Physical addresses refer to hardware addresses of physical memory.
Virtual addresses refer to the virtual store viewed by the process.

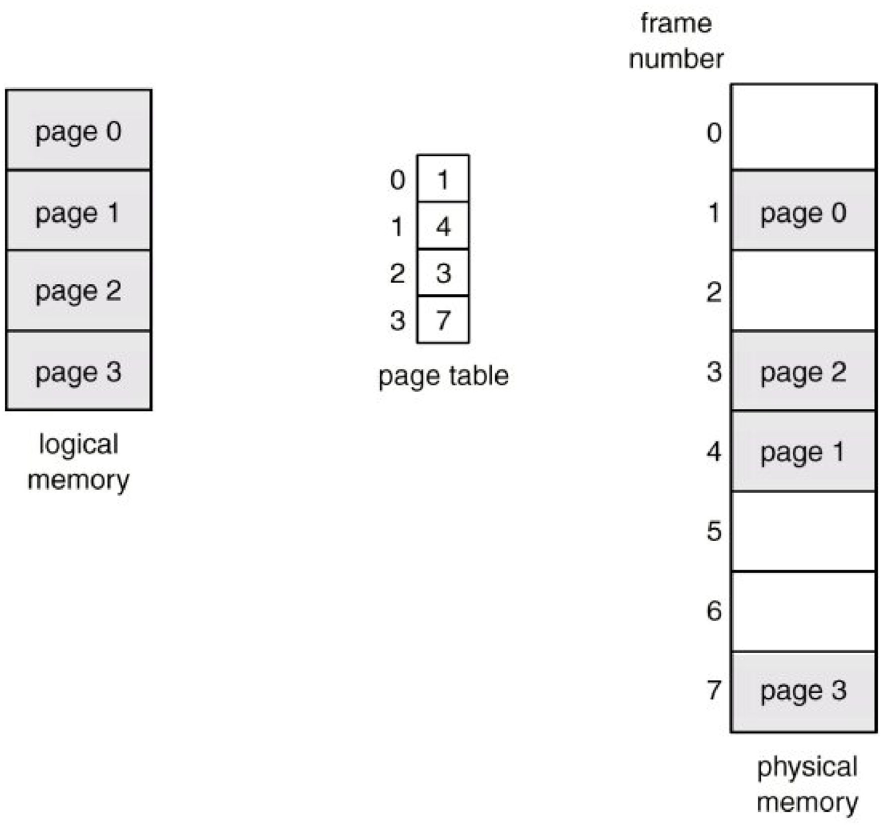
- virtual addresses might be the same as physical addresses
- might be dierent, in which case virtual addresses must be mapped into physical
- addresses. Mapping is done by Memory Management Unit (MMU).
- virtual space is limited by size of virtual addresses (not physical addresses)
- virtual space and physical memory space are independent

Notes:

10.4.2 Paging

Paging is the most common memory management technique:

- virtual space of process divided into fixed-size pages
- virtual address composed of page number and page oset
- physical memory divided into fixed-size frames
- page in virtual space ts into frame in physical memory
- Mapping of VA to PA is done with a page table. Each process has an associated page table.

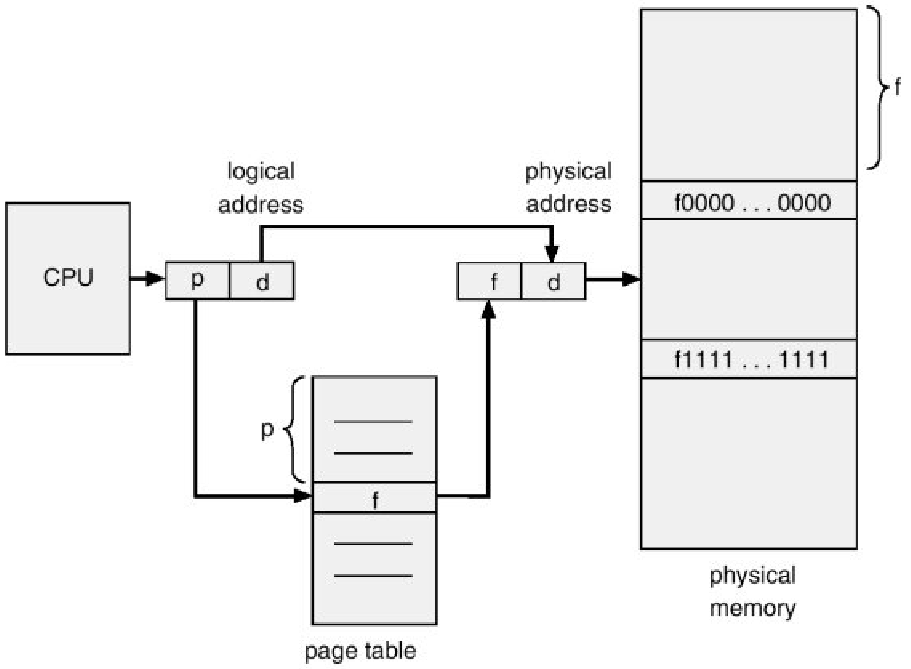


Notes:

10.4.3 Address Translation Architecture

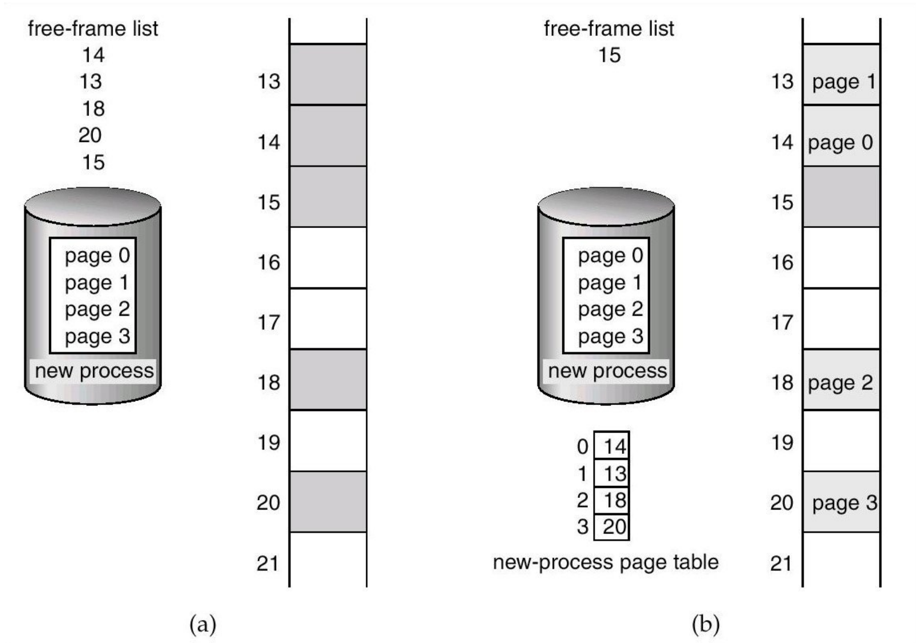
Start execution, causing address translation to be used.

- extract page number from virtual address (show page"|"oset). If we assume 1K page then 1024 bytes then oset is 10 bits.
- if page number is greater than number of pages, generate an illegal page trap
- access appropriate page table entry
- if page is not in memory, trap missing page
- if access violation, trap protection violation (segmentation violation in Unix)
- finally, return physical address (frame number * page size + page offset)



Notes:

10.4.4 Free Frames



Notes:

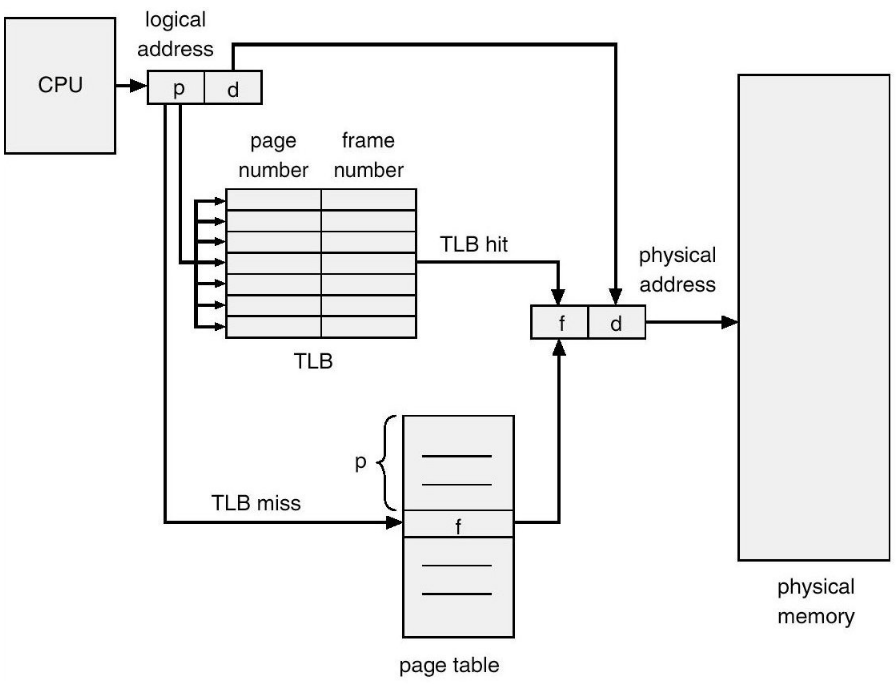
10.4.5 Implementation of Page Table

Page table is kept in main memory. Page-table base register (PTBR) points to the page table. Notes:

10.4.6 Paging Hardware with Page TLB

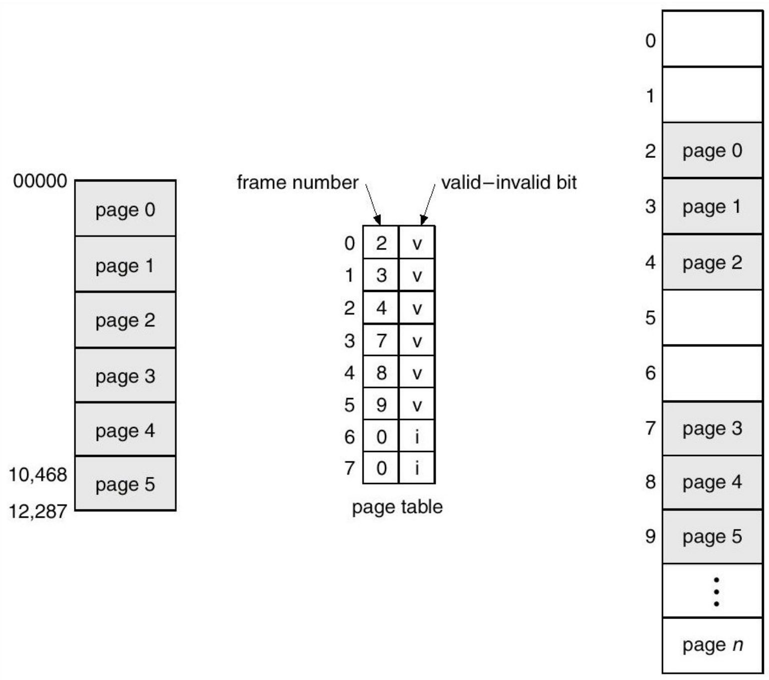
To speed translations, hardware maintains a cache called the translation lookaside buer(TLB). Part of the MMU. Thus, virtual memory accesses:

- check TLB for desired mapping; if present, we are done
- if not present in TLB, consult mapping tables in (slower) memory
- place virtual/physical address pair in cache



Notes:

10.4.7 Memory Protection Scheme



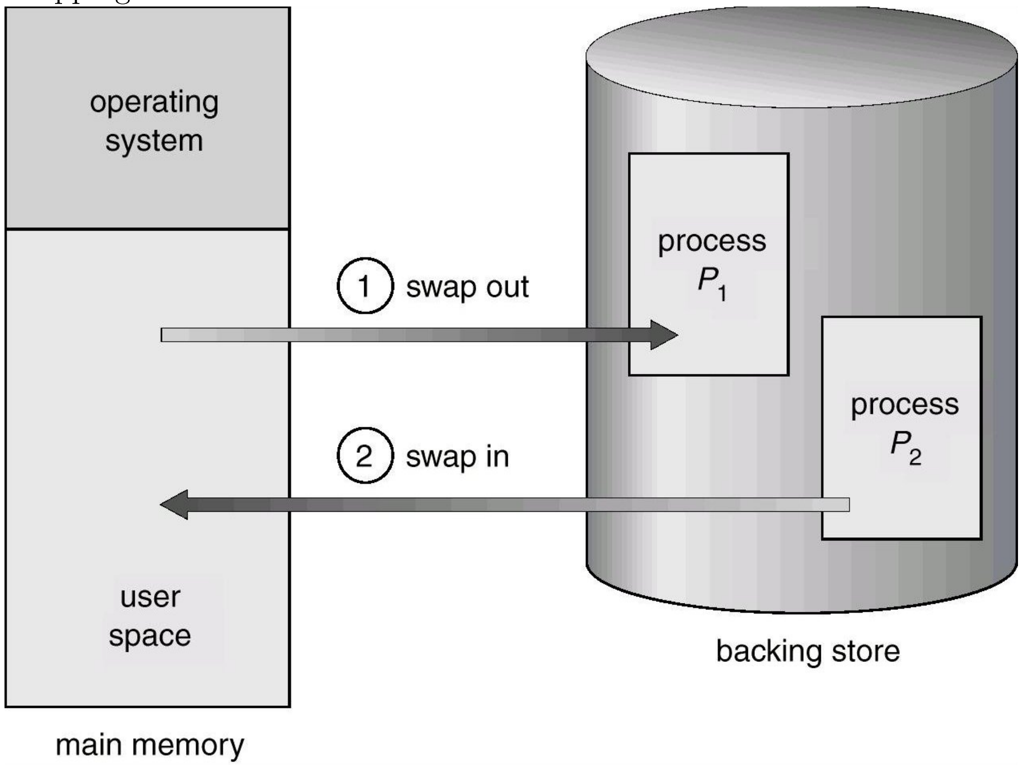
Notes:

10.5 Swapping

If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

The operating system must preserve the contents of that page, if the page is modified, so that it can be accessed at a later time. This type of page is known as a dirty page and when it is removed from memory it is saved in a special sort of file called the swap file.

Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.



Notes:

Appendix A

Assignment Guidelines

The following highlights common deficiencies which lead to loss of marks in Programming assignments. Review this sheet before turningin each Assignementt to make sure that the it is complete in all respects.

A.1 Quality of the Source Code

A.1.1 Variable Names

- Use variable names with a clear meaning in the context of the program whenever possible.

A.1.2 Indentation and Format

- Include adequate white-space in the program to improve readability. Insert blank lines to group sections of code. Use indentation to improve readability of control flow. Avoid confusing use of opening/closing braces.

A.1.3 Internal Comments

- Main program comments should describe overall purpose of the program. You should have a comment at the beginning of each source file describing what that file contains/does. Function comments should describe their purpose and other pertinent information, if any.

- Compound statements (control flow) should be commented. Finally, see that commenting is not overdone and redundant.

A.1.4 Modularity in Design

- Avoid accomplishing too many tasks in one function; use a separate module (Split your code into multiple logical functions). Also, avoid too many lines of code in a single module; create more modules. Design should facilitate individual module testing. Use automatic/local variables instead of external variables whenever possible. Use separate header files and implementation files for unrelated functions.

A.2 Program Performance

A.2.1 Correctness of Output

- Ensure that all outputs are correct. Incorrect outputs can lead to substantial loss in grade

A.2.2 Ease of Use

- The program should facilitate repeated use when used interactively and should allow easy exit. Requests for interactive input from the user should be clear. Incorrect user inputs should be captured and explained. Outputs should be well-formatted.

Appendix B

Grading of Programming Assignments

- Total points per assignment = 10
- Points for timely/early submission = 1
- The source code is out of 3 points. The distribution of points is as follows:
  - (a) The existence of the code itself (1 pts)
  - (b) Proper indentation of the code and comments (1 pts)
  - (c) Proper naming of the functions, variables + Modularity + (1 pts)
- You get 4 points if the program does exactly what it is supposed to do.
- Two (2) points are reserved for the ease of use, the type of user interface, the ability to handle various user input errors, or any extra features that your program might have.